17 | 流量隔离：MongoDB 数据库隔离是怎么做的？

2021-11-25 高楼

《全链路压测实战30讲》 课程介绍 >

讲述：高楼
时长 17:46 大小 16.27M

你好，我是高楼。

这节课，我们详细来讲讲如何基于微服务技术落地 MongoDB 数据库隔离。

快速了解 MongoDB

MongoDB 是目前比较主流的一款 NoSQL 文档数据库，它能够解决关系数据库中的 Schema 问题，其特点是 No-Schema，任意存储和读取任意数据。

MongoDB 的数据格式主要是 JSON/BSON，因为用它新增字段简单，历史数据容错性强，比较容易存储复杂数据，所以很多电商系统都会用它，我们这个开源电商项目里面的会员服务（mall-member）也用到了它。

目前 SpringBoot 对 MongoDB 使用的支持，主要通过 Spring Data MongoDB 模块，有以下两种方式：

MongoTemplate

MongoRepository

既然 Template 和 Repository 都可以做增删查改操作，它们俩的区别和优劣势在哪里呢？

到目前为止，Repository 比 Template 方便得多，但是后者可以提供对执行内容更细粒度的控制。可以说，“方便”和 “功能强大”在某种程度上是相互矛盾的目标。

Template 能做各式各样的操作，比较灵活，但是什么都得自己写。而 Repository 做了很好的封装，Repository 的实现和 JPA 类似，SQL 是动态生成出来的，对应关系是在实体类上定义的，用起来也比较方便，但是对于更复杂的查询操作，最后还是得靠 Template 来实现。

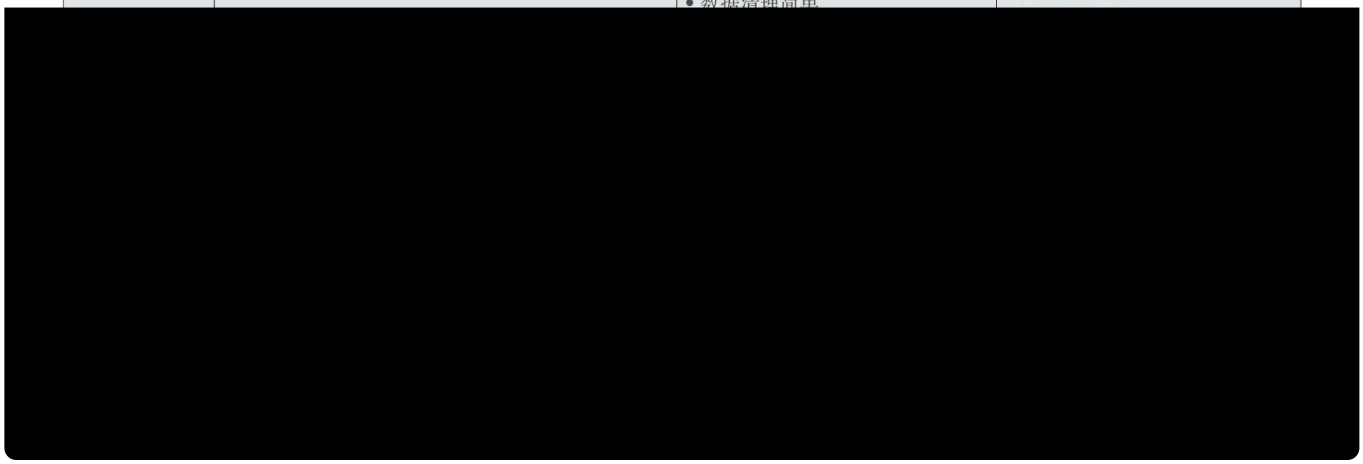
我们这个电商项目里面，主要是使用 MongoTemplate 方式。

数据库隔离方案

刚才，我们简单了解了什么是 MongoDB 以及 SpringBoot 中如何使用 MongoDB，接下来进入这节课的重点，如何在 SpringBoot 中落地数据隔离方案。

首先，我们来回顾一下目前业界对于数据库隔离的主要解决方案，以及它们的优缺点和适用场景。

	数据偏移	影子库	影子表
优点	• 实现简单，无须改造中间件	• 物理隔离 • 对业务数据无侵入性 • 安全性高，对生产库无任何影响 • 扩展性较强 • 数据清理简单	• 逻辑隔离 • 中间件容易支持 • 安全性一般，对生产库有一定影响 • 扩展性较强



毋庸置疑，这里最优最安全的方案是**影子库**。所以在这个项目里，我们选择的 MongoDB 数据库隔离方案还是影子库，也就是说，将压测流量对 MongoDB 的读写打到影子库上。

既然选定了改造方案，我们就梳理一下 MongoDB 影子库改造的需求：

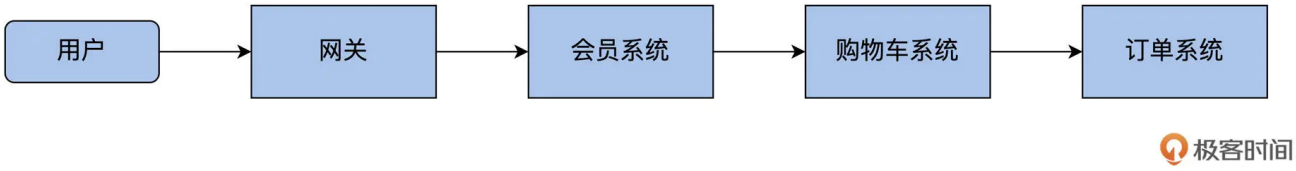
1. 服务能获取压测请求的标记；
2. MongoDB 能支持多数据源；
3. MongoDB 能根据压测标记动态切换数据源；
4. 对业务代码的侵入性尽可能少。

接下来，我们看看如何去落地这些需求。

技术预演

首先，我们写个 demo 尝试做一下技术预演。

这里使用的 demo 工程还是 14 讲的示例，主要包括网关、会员系统、购物车系统和订单系统 4 个服务：



这里我们主要改造会员系统，实现 MongoDB 影子库技术预演。

集成 MongoDB

在 member 工程的 pom 文件引入 MongoDB 依赖：

复制代码

```
<!--引入 mongodb 依赖-->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-mongodb</artifactId>
        </dependency>
```

在 application.yml 中配置 MongoDB 连接信息：

复制代码

```
spring:
  data:
    mongodb:
      shadow: mongodb://localhost:27017/shadow
      produce: mongodb://localhost:27017/produce
```

可以看到，这里配置了多数据源，分别配置 shadow 为影子库，而 produce 为生产库。

实现获取压测标记

接下来我们就要考虑如何获取压测请求 Header 标记了。

首先还是实现一个全局 Filter 过滤器，只要从 Span Baggage 获取压测标记放入 Context（数据上下文），就能在服务中使用了。

全局 Filter 过滤器代码如下：

复制代码

```
package com.dunshan.member.config;

import brave.Span;
import brave.Tracer;
import brave.baggage.BaggageField;
import lombok.extern.log4j.Log4j2;
import org.apache.commons.lang3.StringUtils;
import org.springframework.stereotype.Component;
import javax.servlet.*;
import java.io.IOException;

/**
 * @author dunshan
 * @description: 全局过滤器
 * @date 2021-10-03 17:45:30
 */
@Log4j2
@Component
public class ContextFilter implements Filter {
    private final Tracer tracer;
    ContextFilter(Tracer tracer) {
        this.tracer = tracer;
    }

    @Override
    public void init(FilterConfig filterConfig) throws ServletException {
        Filter.super.init(filterConfig);
    }

    @Override
    public void doFilter(ServletRequest servletRequest, ServletResponse servle
        BaggageField dunshan = BaggageField.getByName("dunshan");
        String flag = dunshan.getValue();
        Span currentSpan = this.tracer.currentSpan();
        AppContext appContext = new AppContext();
        if (StringUtils.isNotBlank(flag)) {
            log.info("压测流量: " + flag);
            currentSpan.tag("flag", flag);
            appContext.setFlag(flag);
            AppContext.setContext(appContext);
        } else {
            log.info("正常流量: " + flag);
            AppContext.setContext(appContext);
        }
        filterChain.doFilter(servletRequest, servletResponse);
    }

```

```
48      @Override
49      public void destroy() {
50          AppContext.removeContext();
51          Filter.super.destroy();
52      }
53  }
```

数据上下文代码如下：

复制代码

```
1  package com.dunshan.member.config;
2
3  import com.alibaba.ttl.TransmittableThreadLocal;
4  import java.io.Serializable;
5
6  /**
7   * @author dunshan
8   * @description: 数据上下文
9   * @date 2021-11-12 17:53:39
10  */
11 public class AppContext implements Serializable {
12     private static final TransmittableThreadLocal<AppContext> contextdunshan =
13     private String flag;
14     public static AppContext getContext() {
15         return contextdunshan.get();
16     }
17     public static void setContext(AppContext context) {
18         contextdunshan.set(context);
19     }
20     public static void removeContext() {
21         contextdunshan.remove();
22     }
23     public String getFlag() {
24         return flag;
25     }
26     public void setFlag(String flag) {
27         this.flag = flag;
28     }
29 }
```

后面，我们只要在业务方通过从 getContext().getFlag() 中取出 Header 标记即可。

代码如下：

复制代码

```
// 获取标记
String header = AppContext.getContext().getFlag();
```

实现动态数据源

要让 MongoDB 实现多数据源，网上其实有很多文章可以参考，原理都是利用 Spring IOC 容器实例化出多个 MongoTemplate。

比如这段代码，一个 Primary 库和一个 Secondary 库，其中 Primary 为主库，在使用的过程中必须指定主库，不然会报错：

复制代码

```
@Configuration
public class MultipleMongoConfig {

  @Autowired
  private MultipleMongoProperties mongoProperties;

  @Primary
  @Bean(name = "primaryMongoTemplate")
  public MongoTemplate primaryMongoTemplate() throws Exception {
    return new MongoTemplate(primaryFactory(this.mongoProperties.getPrimary())
  }

  @Bean
  @Qualifier("secondaryMongoTemplate")
  public MongoTemplate secondaryMongoTemplate() throws Exception {
        return new MongoTemplate(secondaryFactory(this.mongoProperties.getSeco
  }

  @Bean
  @Primary
  public MongoDbFactory primaryFactory(MongoProperties mongo) throws Exception
    MongoClient client = new MongoClient(new MongoClientURI(mongoProperties.ge
    return new SimpleMongoDbFactory(client, mongoProperties.getPrimary().getDa
  }

  @Bean
  public MongoDbFactory secondaryFactory(MongoProperties mongo) throws Excepti
    MongoClient client = new MongoClient(new MongoClientURI(mongoProperties.ge
    return new SimpleMongoDbFactory(client, mongoProperties.getSecondary().get
  }
}

```

复制代码

```
@Configuration
@EnableConfigurationProperties(MultipleMongoProperties.class)
@EnableMongoRepositories(basePackages = "com.dunshan.repository.primary",
    mongoTemplateRef = "primaryMongoTemplate")
public class PrimaryMongoConfig {
}
```

这里我们总结一下主要的实现方式：

> 通过 @Autowired 注入 DataSource 实例，以此操作不同的 Mongo 数据库；
>
> 通过 MongoTemplate 配置的 basePackages，来指定继承了 MongoRepository 的 model 接口，在操作数据库时，使用所配置的 MongoTemplate 实例，以操作不同的数据库。

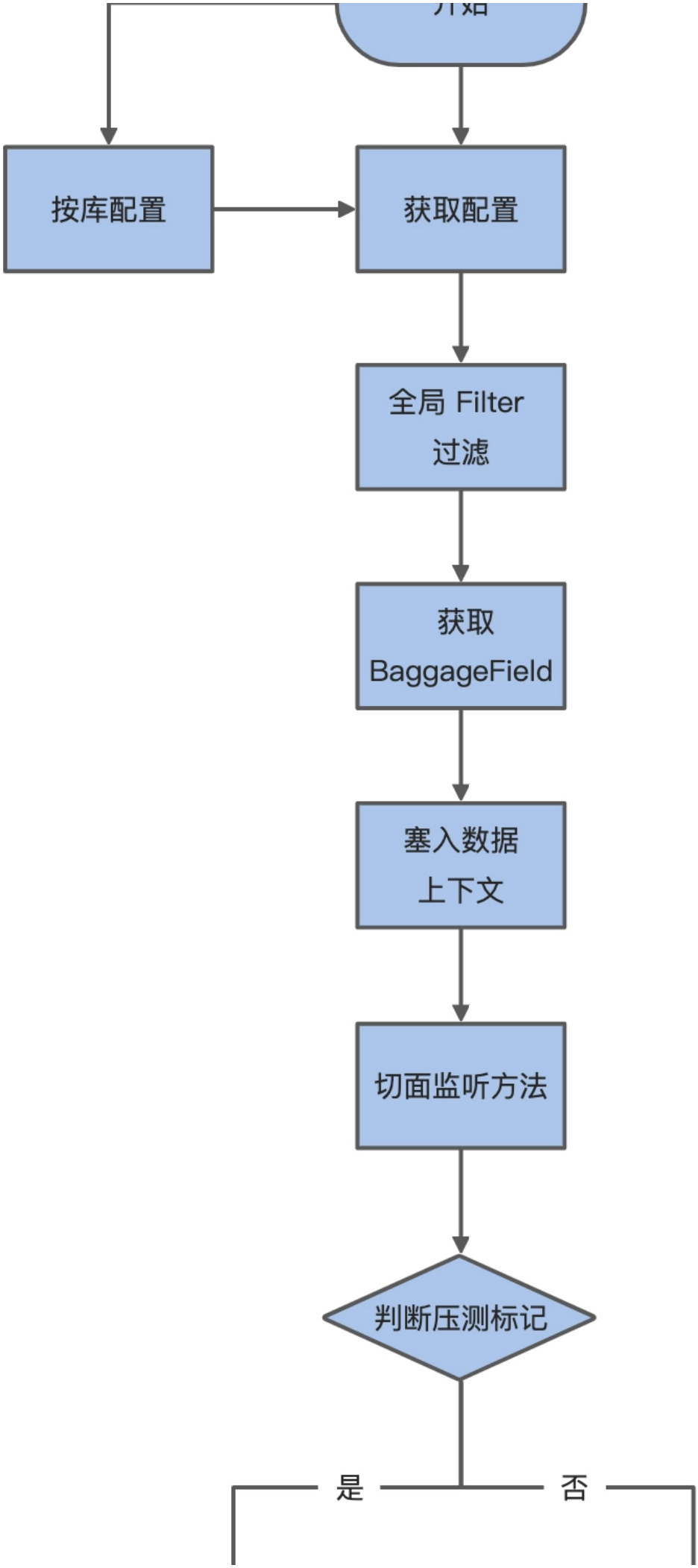
如上所述，这样一层一层注入。

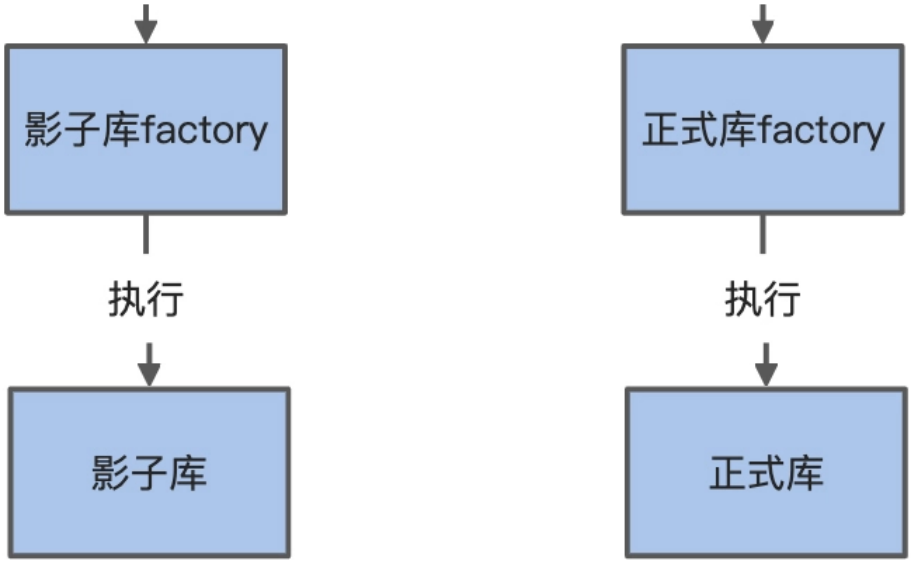
首先，创建 DataSource，然后创建 SimpleMongoDbFactory，最后包装到 MongoTemplate 中。其中需要指定分库的 MongoRepository 以及分库 DAO 层代码。

这是最常见的 SpringBoot Mongo 多数据配置实现方法。这种方式，多数据源确实可用，但不足之处在于，需要根据不同数据源建立不同的 package，一旦数据源发生变更，需要更改所在的 package。另外，这种方式也无法做到动态切换数据源，不是我们想要的方案。

我们可以看到，实现 Mongo 多数据库的核心原理是，需要操作不同的 MongoTemplate 实例。

于是这里我们考虑通过 Spring AOP 技术，在调用方法前后动态替换 SimpleMongoClientDatabaseFactory 值，让每个 factory 都维护自己需要连接的库。在操作之前，先判断请求 Header 标记，然后选择对应 factory，切换完成后再放回去，也就是说将相关数据源的 Template 删除。


开始
按库配置
获取配置
全局 Filter
过滤
获取
BaggageField
塞入数据
上下文
切面监听方法
判断压测标记
是
否



极客时间

具体怎么操作呢？

首先，添加 AOP 和 TransmittableThreadLocal 依赖：

复制代码

```
    <!--引入 AOP 依赖-->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-aop</artifactId>
        </dependency>
        <!-- 引入 TransmittableThreadLocal 依赖-->
        <dependency>
            <groupId>com.alibaba</groupId>
            <artifactId>transmittable-thread-local</artifactId>
            <version>2.12.1</version>
            <scope>compile</scope>
        </dependency>
```

接着创建 AOP 切面：

复制代码

```
package com.dunshan.member.config;

import lombok.extern.slf4j.Slf4j;
import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.Around;
```

```
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Pointcut;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.data.mongodb.core.MongoTemplate;
import org.springframework.data.mongodb.core.SimpleMongoClientDatabaseFactory;
import org.springframework.stereotype.Component;
import java.lang.reflect.Field;

/**
 * Created by dunshan on 2021/8/29.
 *
 * @return
 */

@Component
@Aspect
@Slf4j
public class MongoSwitch {

    //获取配置文件的副本集连接
    @Value("${spring.data.mongodb.produce}")
    private String produceUri;

    @Value("${spring.data.mongodb.shadow}")
    private String shadowUri;

    @Pointcut("execution(public * com.dunsan.member.service.MongotemplteServic
    public void routeMongoDB() {
    }

    @Around("routeMongoDB()")
    public Object routeMongoDB(ProceedingJoinPoint joinPoint) {
        Object result = null;
        Object o = joinPoint.getTarget();
        Field[] fields = o.getClass().getDeclaredFields();
        MultiMongoTemplate mongoTemplate = null;

        // 获取标记
        String header = AppContext.getContext().getFlag();

        try {
            for (Field field : fields) {
                field.setAccessible(true);
                Class fieldclass = field.getType();

                // 找到 Template 的变量
                if (fieldclass == MongoTemplate.class || fieldclass == MultiMo
                    // 查找项目对应的 MongFactory
                    SimpleMongoClientDatabaseFactory simpleMongoClientDbFactor

                    // 实例化
                    if (header != null && header.equals("7DGroup")) {
```

```
                    simpleMongoClientDbFactory =
                            new SimpleMongoClientDatabaseFactory(this.shad
                } else {
                    simpleMongoClientDbFactory =
                            new SimpleMongoClientDatabaseFactory(this.prod
                }

                // 如果第一次，赋值成自定义的 MongoTemplate 子类
                if (fieldclass == MongoTemplate.class) {
                    mongoTemplate = new MultiMongoTemplate(simpleMongoClie
                } else if (fieldclass == MultiMongoTemplate.class) {
                    Object fieldObject = field.get(o);
                    mongoTemplate = (MultiMongoTemplate) fieldObject;
                }
                // 设置 MongoFactory
                mongoTemplate.setMongoDbFactory(simpleMongoClientDbFactory
                // 重新赋值
                field.set(o, mongoTemplate);
                break;
            }
        }
        try {
            result = joinPoint.proceed();
            // 清理 ThreadLocal 的变量
            mongoTemplate.removeMongoDbFactory();
        } catch (Throwable t) {
            log.error("", t);
            mongoTemplate.removeMongoDbFactory();
        }
    } catch (Exception e) {
        log.error("", e);
    }

    return result;
  }
}
```

这里，我再补充一下 Spring AOP 和 AspectJ 的关系。

AspectJ 也是 Spring 的 AOP 实现方式之一，可以用 CGLIB 对类做动态代理，也可以用 Java 自己的基于接口的动态代理，也可以用 AspectJ。Spring 提供了两个有关 AOP 的包，一个是 spring-aop，一个是 spring-aspect。

然后，我们创建相关配置类:

复制代码

```java
package com.dunshan.member.config;

import com.mongodb.client.MongoDatabase;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.data.mongodb.core.MongoDatabaseFactorySupport;
import org.springframework.data.mongodb.core.MongoTemplate;

/**
 * Created by dunshan on 2021/8/29.
 *
 * @return
 */

public class MultiMongoTemplate extends MongoTemplate {

    //用来缓存当前 MongoDbFactory
    private static ThreadLocal<MongoDatabaseFactorySupport> mongoDbFactoryThre

    public MultiMongoTemplate(MongoDatabaseFactorySupport mongoDbFactory) {
        super(mongoDbFactory);
        if (mongoDbFactoryThreadLocal == null) {
            mongoDbFactoryThreadLocal = new ThreadLocal<>();
        }
    }

    public void setMongoDbFactory(MongoDatabaseFactorySupport factory) {
        mongoDbFactoryThreadLocal.set(factory);
    }

    public void removeMongoDbFactory() {
        mongoDbFactoryThreadLocal.remove();
    }

    @Override
    public MongoDatabase getDb() {
        return mongoDbFactoryThreadLocal.get().getMongoDatabase();
    }
}
```

这里主要是为了实现多线程下存储 MongoDbFactory 副本变量的，也就是 MongoDbFactory 的上下文。

最后，我们需要写一个简单的接口测试一下。

新建一个实体类 Student：

复制代码

```
package com.dunshan.member.entity;

import lombok.Data;
import org.springframework.data.mongodb.core.mapping.Document;
/**
 * Created by dunshan on 2020/8/29.
 *
 * @return
 */

@Data
@Document("student")
public class Student {
    private String name;
    private String addr;
    private String email;
}

```

再新建 MongotemplteService：

复制代码

```
package com.dunshan.member.service;

import org.springframework.data.mongodb.core.MongoTemplate;
import org.springframework.stereotype.Service;

/**
 * Created by dunshan on 2021/8/29.
 *
 * @return
 */

@Service
public class MongotemplteService {
    private MongoTemplate mongoTemplate;

    public <T> T insert(T var1) {
        return mongoTemplate.insert(var1);
    }
}

```

这里，我们实现了一个 insert 方法。

接下来，新建控制器 HomeController：

复制代码

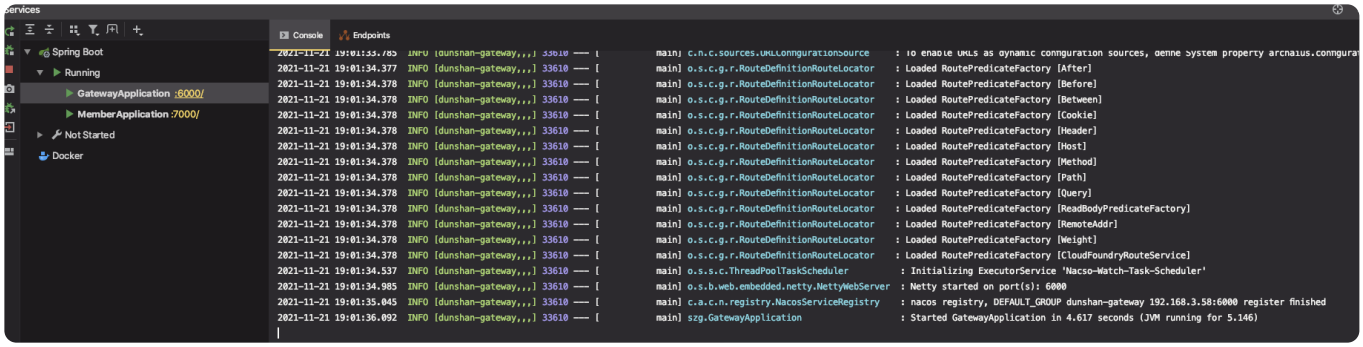
```
package com.dunshan.member.api;

import com.dunshan.member.service.MongotemplteService;
import com.dunshan.member.entity.Student;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.PutMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

@RequestMapping("/home")
@RestController
public class HomeController {

    @Autowired
    private MongotemplteService mongotemplteService;

    @PutMapping
    public String addDataByDynamic(@RequestParam(value = "name") String name,@
        Student student = new Student();
        student.setAddr(addr);
        student.setName(name);
        student.setEmail(email);
        mongotemplteService.insert(student);
        return "添加成功";
    }
}
```

这样，我们就创建了一个添加 student 的接口。

好了，接下来我们验证一下改造效果。

接口测试

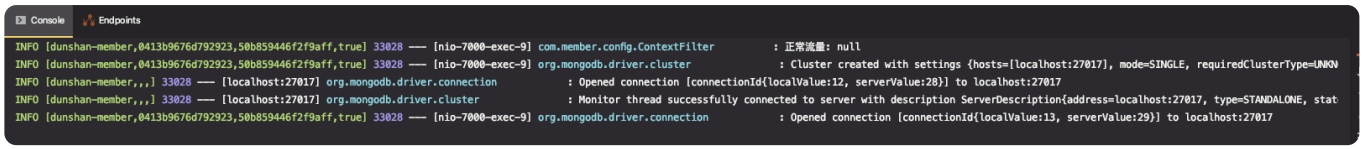
启动网关和会员服务，我们再用 Postman 验证一下这个 demo。



正常流量（不带 Header 压测标记）

首先，我们测试请求接口不带 Header 压测标记的情况：

我们可以看到对应的服务端日志：



这里，数据已经正常落到正式库里面了：

压测流量（带 Header 压测标记）

我们再来测试一下接口带 Header 压测标记的情况：

下面是对应的服务端日志：

我们可以看到，数据已经正常落到影子库里面了，这说明数据已经隔离成功了。

到这里，我们就已经初步实现对于影子库的需求了。

我再总结下 demo 预演的主要步骤：

1. 实现多数据源；
2. 通过全局 Filter 获取标记，并存入数据上下文；
3. 通过 AOP 拦截请求，并判断压测标记；
4. 选择对应 Factory；
5. 操作对应的 MongoDB。

可以看到，这里面的改造量还是挺大的，那么，我们如何把它在真实系统落地呢？

真实系统改造

首先，我们需要梳理一下 mall 电商项目里 MongoDB 的使用方式。

通过仔细阅读项目代码，我们发现只有在 member（会员服务）中会使用 MongoDB ，这里我们以 【添加品牌关注】接口为例，梳理一下代码的实现逻辑。

下面是控制器的代码。

复制代码

```
    @ApiOperation("添加品牌关注")
    @RequestMapping(value = "/add", method = RequestMethod.POST)
    @ResponseBody
    public CommonResult add(@RequestBody MemberBrandAttention memberBrandAtten
        int count = memberAttentionService.add(memberBrandAttention);
        if(count>0){
            return CommonResult.success(count);
        }else{
            return CommonResult.failed();
        }
    }
```

接口逻辑处理如下：

这个逻辑比较简单，主要是通过注入 memberAttentionService 中的 DAO 接口，通过实现 MemberAttentionServiceImpl 去具体操作数据库。

紧接着，我们来看下 MemberAttentionServiceImpl 的代码：

复制代码

```
@Service("MemberAttent")
public class MemberAttentionServiceImpl implements MemberAttentionService {

    @Autowired
    private MemberService memberService;

    @Autowired
    private MongoTemplate mongoTemplate;

    @Override
    public int add(MemberBrandAttention memberBrandAttention) {
        int count = 0;
        UmsMember member = memberService.getCurrentMember();
        memberBrandAttention.setMemberId(member.getId());
        memberBrandAttention.setMemberNickname(member.getNickname());
        memberBrandAttention.setMemberIcon(member.getIcon());
        memberBrandAttention.setCreateTime(new Date());

        Query query = new Query(Criteria.where("memberId").is(memberBrandAtten
        List<MemberBrandAttention> findAttention = mongoTemplate.find(query,Me

        if (findAttention.isEmpty()) {
            mongoTemplate.save(memberBrandAttention);
            count = 1;
        }
        return count;
    }

    ......
}
```

可以看到，我们主要还是通过 MongoTemplate 的方式实现了一个 save 的操作。

到这里就可以确定， mall 项目操作数据库的方式与我们在 demo 的逻辑是一致的了。

通过 demo 的例子，我们发现需要改动的代码还是不少的。那么问题来了，如果做真实系统改造，为了减少原有服务代码的改动，能不能把这些影子库配置、动态切换的逻辑抽取做成一个 SDK，在压测的时候项目直接引入 SDK 依赖呢？

带着这个问题，我研究了一下 SpringBoot 的特性，发现还真有可能，这就要说到 Starter 了。

> SpringBoot 中的 Starter 是一种非常重要的机制，能够抛弃以前繁杂的配置，将其统一集成进 Starter，应用者只需要在 Maven 中引入 Starter 依赖，SpringBoot 就能自动扫描到要加载的信息并启动相应的默认配置。Starter 让我们不用再处理各种依赖库、配置各种信息。SpringBoot 会自动通过 classpath 路径下的类发现需要的 Bean，并注册进 IOC 容器。SpringBoot 提供了针对日常企业应用研发各种场景的 spring-boot-starter 依赖模块。所有这些依赖模块都遵循着约定成俗的默认配置，并允许我们调整这些配置，即遵循“**约定大于配置**”的理念。

既然 SpringBoot 原生就支持 Starter 机制，那么我们就得考虑如何实现影子库的自定义 Starter 功能了。

实现影子库 Starter

创建一个 Maven 项目：dynamicmongo-starter。

将 demo 中的以下文件拷贝到项目中：

注意，我们需要修改 AOP 切面拦截点到对应真实系统的位置。

创建自动装配文件：

复制代码

```
package com.dunshan.dynamicmongo.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

/**
 * @author dunshan
 * @date 2021/8/29 mongo 自动配置
 */

@Configuration(proxyBeanMethods = false)
public class MongodbAutoConfiguration {


```

```
15    @Bean
16    public MongoSwitch mongoSwitch() {
17      return new MongoSwitch();
18    }
19
20    @Bean
21    public MongotemplteService mongotemplteService() {
22      return new MongotemplteService();
23    }
24
  }
```

这里，我们将 MongoSwitch 和 MongotemplteService 类定义为 Bean，交给 IOC 容器。

在 resources 下新建 META-INF 文件夹，然后创建 spring.factories 文件。

复制代码

```
1 org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
2     com.dunshan.dynamicmongo.config.MongodbAutoConfiguration
```

至此，Starter 就已经写完了，是不是很简单？

接下来我们通过 maven install 命令操作把 Starter 的包打入 maven local repository（maven 本地仓库）里面。

最后，我们总结一下自定义 Starter，我们要做的事情主要是两个：

确定依赖；

编写自动配置。

这里重点要做的就是编写自动配置，特别是注解配置的使用，它包括：

@Configuration：指定这个类是一个配置类；

@ConditionalOnXXX：在指定条件成立的情况下自动配置类生效；

@AutoConfigureAfter：指定自动配置类的顺序；

@Bean：给容器中添加组件；

@ConfigurationPropertie：结合相关 xxxProperties 类来绑定相关的配置；

@EnableConfigurationProperties：让 xxxProperties 生效加入到容器中。

按照这些注解写好自动配置类后，我们还需要进行自动配置的加载，加载方式是将需要启动就加载的自动配置类，配置在 META-INF/spring.factories 里。

真实项目引入 Starter

在 member 项目中引入该依赖：

复制代码

```
1       <dependency>
2             <groupId>com.dunshan</groupId>
3             <artifactId>dynamicmongo-starter</artifactId>
4             <version>0.0.1-SNAPSHOT</version>
5         </dependency>
```

接下来配置 MongoDB 连接信息，在 application.yml 中配置：

复制代码

```
1 spring:
2   data:
3     mongodb:
4       produce: mongodb://localhost:27017/mall-port
5       shadow: mongodb://localhost:27017/mall-port_shadow
```

这里分开配置 mall-port_shadow 为影子库，mall-port 为生产库。

接口测试

上面我们已经对系统进行了改造，接下来我们以 member （会员服务）中的 【添加品牌关注】 接口来做下测试，验证影子库功能是否达到了我们的预期。

下面这张图是接口文档中对该接口的描述：

这是一个 Post HTTP 接口，在测试前我们首先需要获取 token。

正常流量（不带 Header 压测标记）

在接口管理平台中测试正常接口请求：

我们可以看到，正常请求的数据已经成功落在正常库里面了。

压测流量（带 Header 压测标记）

接下来，我们通过 curl 命令测试一下，带入 Header 压测标记请求接口：

复制代码

```
~ ▶ curl -X POST -H  "Accept:*/*" -H  "Authorization:eyJhbGciOiJSUzI1NiIsInR5c

{"code":200,"message":"操作成功","data":1}%
```

我们可以看到， Header 带压测标记的请求数据已经成功落在影子库里了：

到这里，我们的真实系统 MongoDB 影子库功能就改造成功了。

总结

好了，这节课就讲到这里，我们使用 demo 预演和系统改造，完整演示了 MongoDB 数据库隔离技术方案的改造过程。这节课有两个要点，我希望你能够记住：

1、MongoDB 数据隔离技术本质上跟 MySQL 是一样的；

2、对于常见的 SDK 注入方案，除了 AOP 字节码增强，我们还可以考虑 SpringBoot 中基于 IOC 的 Starter 机制。

最后，再次强调一遍，在全链路压测中，**数据库隔离是最重要的一个改造环节了**。因为做得不到位，压测流量就会污染生产数据，导致的后果非常严重。这节课给出的影子库方案，是改造成本低，效果最好的一种方式。希望你能够充分地理解并用好它。

课后题

学完这节课，我想请你思考两个问题：

1. 你知道 Spring 中 AOP 和 IOC 的区别是什么？
2. 相比字节码 AOP 增强方案，你觉得 Starter 的优缺点是什么？

欢迎你在留言区与我交流讨论。你也可以把这节课分享给你身边的朋友，他们的一些想法或许会让你有更大的收获。我们下节课见！

分享给需要的人，Ta订阅后你可得 20 元现金奖励

生成海报并分享

赞 1 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 16 | 流量隔离：Redis 缓存隔离是怎么做的？

下一篇 18 | 流量隔离：RabbitMQ 消息隔离是怎么做的？

更多学习推荐

精选留言

写留言

> 由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。